Planning Your Device Driver

Windows Embedded Compact 7 Technical Article

Writers: Mark McLemore

Technical Reviewer: KS Huang, Michael Svob

Published: March 2011

Applies To: Windows Embedded Compact 7

Abstract

This article helps you get started in creating a new device driver for Windows Embedded Compact 7. By reading this article you will gain an essential understanding of the Windows Embedded Compact device driver architecture while you learn how to choose the implementation options that best serve your requirements. After you have read this article, you will have the background knowledge to begin the development and testing phases of your device driver project as described in the companion articles *Implementing Your Device Driver* and *Building and Testing Your Device Driver*.

Introduction

Windows Embedded Compact 7 includes a set of standard device drivers for each board support package (BSP) that it supports. However, you may find it necessary to write a new device driver or to port an existing device driver to support a particular hardware device on your target platform. Microsoft provides many resources to support the device driver development process, including numerous sample device drivers, reference documentation, and debugging tools for use both on the Windows Compact Embedded 7 platform and on your development computer. In addition, we provide a set of technical articles that guide you through the device driver development process.

This article is the first in a series of three articles. *Planning Your Device Driver* walks you through the initial design phase of your device driver project while explaining the most important functionality of the Windows Embedded Compact 7 device driver architecture. *Implementing Your Device Driver* (http://go.microsoft.com/fwlink/?LinkID=210237) helps you begin the development of your device driver and explains how to add functionality to support your device hardware. *Building and Testing Your Device Driver* (http://go.microsoft.com/fwlink/?LinkID=210199) describes the steps for building and debugging your device driver after you have completed your initial implementation. This document, *Planning Your Device Driver,* will help you to become familiar with the Windows Embedded Compact device driver architecture, and help you to select the correct options for your device driver before you begin its implementation.

Your Device Driver

Device drivers are software components that communicate with physical or virtual devices. Device drivers control hardware peripherals such as network adapters, display adapters, mouse devices, keyboards, serial ports, and storage devices. Some device drivers control hardware buses that connect peripheral devices to your target platform, while others control virtual devices (such as a file system) instead of hardware peripherals.

Each device driver encapsulates and abstracts the functionality of its underlying hardware, and makes this functionality available to the OS and applications through an API. This makes it possible for the OS and applications to use the hardware without consideration for the hardware implementation. Nearly every device driver satisfies this general definition, but the specific character and internal definition of your device driver can vary greatly from other device drivers, depending on the following factors:

- The complexity of the hardware that your device driver supports.
- How your device driver interfaces to the OS and applications.
- Whether your device driver implementation reuses existing functionality.
- Where your device driver runs—in kernel memory space or user memory space.

These factors depend, in turn, on your product requirements and your development priorities. Windows Embedded Compact 7 offers numerous options to address the needs of different platforms, diverse hardware devices, and varying product requirements. To start a device driver project that is tailored specifically to your device hardware and your target platform, first you must understand which of these options are relevant to your requirements, and then select the options that are the most appropriate for your device driver needs. With your device driver options defined, you can concentrate only on the Windows Embedded Compact 7 functionality that is relevant to your project.

Characterizing Your Device Driver

Before selecting from the various device driver options provided by Windows Embedded Compact 7, determine your device driver's required characteristics, and then select the interface type, device driver model, run environment, loading mechanism, and access method that offer these characteristics. Use the following steps to determine your device driver's requirements.

Step 1 – Identify the interface between your device driver and the device.

If your device driver communicates directly with device hardware, this interface consists of the memory ranges, registers, and/or ports that the device driver uses to control and move data to and from the device. An example of this type of driver is a display driver. If your device driver communicates with its device through an intermediate device (such as a bus), this interface is whatever application programming interface (API) that is exposed by the intermediate device. An example of this type of driver is a driver that communicates with a device through a USB connection.

Step 2 – Identify the APIs that the device driver must expose.

If the device driver is intended to provide services to a specific application or a component of the operating system that already has a defined interface, your device driver must expose that interface. For example, if you are writing a new audio device driver, your device driver must expose the same interface as that exposed by other audio device drivers in the system. If your driver has no predefined interface to expose to the operating system or to applications, you must design an "upper interface" (to the OS and applications) for your driver. In this case, we encourage you to use the stream interface. For more on the stream interface, see the section "Stream Drivers" later in this document.

Step 3 – Select a device driver interface type that supports the APIs that your device driver must expose.

This can be a stream interface or a native interface. For more information about device driver interface types, see the section "Stream Drivers and Native Drivers" later in this document.

Step 4 – Determine an appropriate mapping between the upper and lower interfaces.

This maps the application-level interface (the functions that applications or the operating system call in the device driver) to the functionality provided by the device. For example, an application-level **ReadFile** call or an **XXX_Read** call into a storage device driver typically maps to a block-read operation on a storage device.

Step 5 – Identify any additional architectural factors that are relevant to your device driver.

For example, consider the following:

- Whether your driver must be componentized into separate, reusable modules.

- What parts of your driver, if any, can run outside the kernel for enhanced system robustness or security.
- Which functions are likely to have the most time or memory constraints.

The section “Layered and Monolithic Drivers” later in this document provides more information on choosing an appropriate architectural model for your device driver.

Step 6 – Identify any remaining requirements and tradeoffs that go into the design of your device driver.

For example, consider the following questions:

- Must your device driver emphasize performance over resource optimization?
- Are you reusing code from an existing device driver to speed your development time?
- Is development time more important than performance, CPU usage, and memory footprint?
- Can your device be powered on and off, and can it be power-managed?

Each section in the remainder of this article describes the tradeoffs between performance, robustness, security, resource usage, and ease of development for each option. When you have completed the steps listed earlier in this section, and as you read through the rest of this article, you can make informed choices from among the many device driver design options based on this preliminary characterization of your device driver.

Choosing Device Driver Types

In Windows Embedded Compact 7, device drivers are DLLs that contain hardware-specific code. However, not all device drivers are used or categorized in the same way. Windows Embedded Compact 7 classifies device drivers using several different perspectives:

- Which interface the device driver presents to the OS and applications.
- How the device driver is structured internally.
- Whether the device driver runs in the kernel or in user space.
- Whether the driver manages a bus.

Within each of these perspectives, there are different device driver types that you can select from. Your device driver will exhibit some combination of these driver types, depending on how it interfaces with other software components (*stream* or *native* device driver), how it is structured internally (*monolithic* or *layered* device driver), which address space it runs in (*user-mode* or *kernel-mode*), and whether it manages a bus.

Stream Drivers and Native Drivers

In Windows Embedded Compact 7, your driver will be either a *stream driver*, and interface with the OS and applications through the *stream* application programming interface (API), or a *native driver,* and interface with the OS through a *native* interface. Native and stream drivers differ only in terms of the interfaces they expose and not by the devices they control. You can load both types of drivers during system startup or on demand, and either can be structured as layered or monolithic drivers (see “Layered and Monolithic Drivers" later in this article). Which interface you select for your device driver depends on the purpose of your device driver and how the OS and applications access your device.

A different system software component manages each driver interface type. The *Device Manager* is the software component that manages stream device drivers on the system; it loads, unloads, and interfaces with stream drivers (see "The Device Manager" later in this article). The Graphics, Windowing, and Events Subsystem (GWES) loads and manages most native drivers. As shown in the following figure, applications access stream drivers through the Device Manager while the OS accesses both types of drivers through the GWES and the Device Manager.

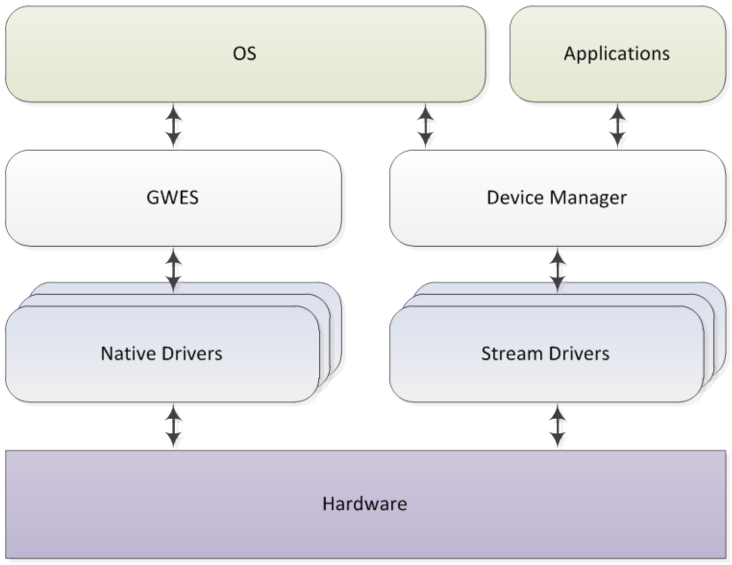

Figure 1 - Stream drivers and native drivers

The majority of device drivers in Windows Embedded Compact 7 are stream drivers. The following table summarizes key distinctions between stream drivers and native drivers.

Table 1 - Device Driver Interface Types

Type	**Characteristics**
Stream	• Supports any device whose I/O operations are similar to the operation of reading and writing to a file. • Exposes the stream interface functions. • Loaded by the Device Manager at system boot time, device detection time, or application load time. • Exposed through the file system.

Type	Characteristics
Native	• Supports any device. • Usually loaded and called by the core OS at boot time. • Exposes a custom API set that is unique to the driver. • Typically supports input and output peripherals such as display drivers, keyboard drivers, and touchscreen drivers.

Stream Drivers

Windows Embedded Compact 7 includes the stream interface API, which makes it possible for software modules to interact with peripherals as if they were files. A stream driver (sometimes referred to as a "stream interface driver") is any driver that exposes the stream interface, regardless of the type of the device that the driver controls. The stream interface is appropriate for any I/O device that can be considered a data source or a data sink.

Stream Interface API

The stream interface functions are designed to closely match the semantics of file system APIs such as **ReadFile**, **WriteFile**, and **IOControl**. Device functionality presented through the stream interface is exposed to applications through the file system; that is, applications interact with the driver by opening specially named files in the file system.

The stream interface consists of thirteen functions as shown in the following table.

Table 2 - Stream Interface Functions

Function Name	Description
XXX_Init	Initializes the device.
XXX_Open	Opens the device.
XXX_Close	Closes the device.
XXX_Read	Reads data from the device.
XXX_Write	Writes data to the device.
XXX_Seek	Moves the data pointer in the device.
XXX_IOControl	Sends a command to the device.
XXX_Cancel	Cancels all pending asynchronous I/O requests.
XXX_Deinit	De-initializes the device.
XXX_PreDeinit	Prepares the device driver for de-initialization.
XXX_PreClose	Prepares the device driver for a close operation.

Function Name	**Description**
XXX_PowerDown	Ends power to the device.
XXX_PowerUp	Restores power to the device.

The ***XXX*** prefix is a placeholder: you replace ***XXX*** with a prefix appropriate for your specific device driver implementation. Alternately, you can use undecorated entry point names that do not have the ***XXX*** prefix. For more about stream interface functions, see Stream Interface Driver Reference (http://go.microsoft.com/fwlink/?LinkID=210285) in Windows Embedded Compact 7 Documentation.

When to Create a Stream Driver

Consider using a stream driver design for your device driver if any of the following are true:

- Your device driver manages a peripheral that produces or consumes streams of data as its primary function. For example, applications typically access storage, serial port, and audio devices through stream drivers. An example of a device that is not a good candidate for a stream driver is a display device, because it does not produce or consume data using traditional stream-oriented file system methods.
- Applications that use your device driver are currently implemented to access device functionality through a file system interface.
- You are beginning your device driver development with an existing Windows Embedded Compact 7 sample driver that is a stream driver.
- You are porting your device driver from another operating system. Because the stream interface approach is common to many operating systems, you may find it easiest to port an existing device driver to Windows Embedded Compact 7 by adapting it to the stream interface.

Sample Stream Driver

Microsoft provides source code for a sample stream driver at %_WINCEROOT%\platform\BSPTemplate\src\drivers\streamdriver. See the companion article *Implementing Your Device Driver* for more information on using this sample driver to develop a new device driver.

For More Information About Stream Drivers

Windows Embedded Compact 7 Documentation includes detailed reference material for the design and implementation of stream interface drivers. If you are creating a stream driver, be sure to read the following:

Table 3 - Stream Driver Topics

Topic	**Description**
Stream Interface Driver Functions	Explains each of the stream interface driver functions, describing syntax, parameters, and return values.

Topic	Description
Device Manager Functions	Device Manager functions for loading stream drivers, issuing I/O controls to devices, and handling asynchronous read, write, and I/O control operations.
File I/O Functions	File system functions that applications use to access stream devices, including **CreateFile**, **ReadFile**, and **WriteFile**.

Native Drivers

Native drivers are drivers that expose any API set other than the stream interface functions. Unlike stream drivers, each native driver implements a specific function according to its purpose. Because the device driver functions that you implement for a native driver are specific to the functionality of the hardware and the way that functionality is used by subsystems such as GWES, there is no common set of interface functions for native drivers to expose, as in the case of stream drivers. Beyond this difference, native drivers behave exactly like stream drivers. They can be structured in a monolithic or layered design (see the section “Layered and Monolithic Drivers” later in this document), they can manage devices that are peripheral or built-in to the platform, and they can be loaded either when the system boots or on demand, such as when the user connects a device to the platform.

When to Create a Native Driver

Consider using a native driver design if any of the following are true:

- Your device is a display device or user input device.
- Your device is controlled by GWES.
- You have special device access requirements that do not fit into the stream interface model.
- You are beginning your device driver development with an existing Windows Embedded Compact 7 sample driver that is a native driver.

Sample Native Drivers

Microsoft has pre-designed the APIs exposed by device drivers that interact with certain portions of the Windows Embedded Compact 7 operating system, such as GWES. Because we provide sample code for these drivers, you typically only need to port the appropriate sample driver to your specific hardware, and do not need to define the API set that your driver will expose. The provided native sample drivers include:

- Sample Display Driver
- Sample Keyboard Driver

The source code for each of these sample drivers is located at %_WINCEROOT%\public\common\oak\drivers

These sample native device drivers present a standard set of functionality for all devices of a particular class, and all representatives of a specific class conform to the same interface. The Windows Embedded Compact operating system can then treat all instances of a particular device class in a similar way. For example, the OS uses both

UHCI- and OHCI-compliant USB controller hardware in the same way, because the USB host controller driver hides the differences between these hardware implementations from the operating system, even though those two types of USB controller hardware are internally very different.

For More Information About Native Drivers

If you are creating a native driver, you are likely creating a display driver or a user interface driver. The following Windows Embedded Compact 7 Documentation explains these types of drivers in detail:

Table 4 - Native Driver Topics

Topic	Description
Display Drivers	Describes the native device driver interface between GWES and display drivers.
Touch Screen Drivers	Describes the interface for devices that users can touch with either a stylus or their fingers to provide user input.
Keyboard and Mouse Drivers	Describes the interface for Windows Embedded Compact keyboard drivers, which also encapsulate mouse driver functionality.

Layered and Monolithic Drivers

You can implement your device driver as a *layered* driver or a *monolithic* driver. Layered drivers, also known as *MDD/PDD* drivers, split interface and hardware-specific functionality into two separate components: a model device driver (MDD) layer that contains the interface to the operating system and applications, and a platform device driver (PDD) layer that communicates with the underlying hardware.

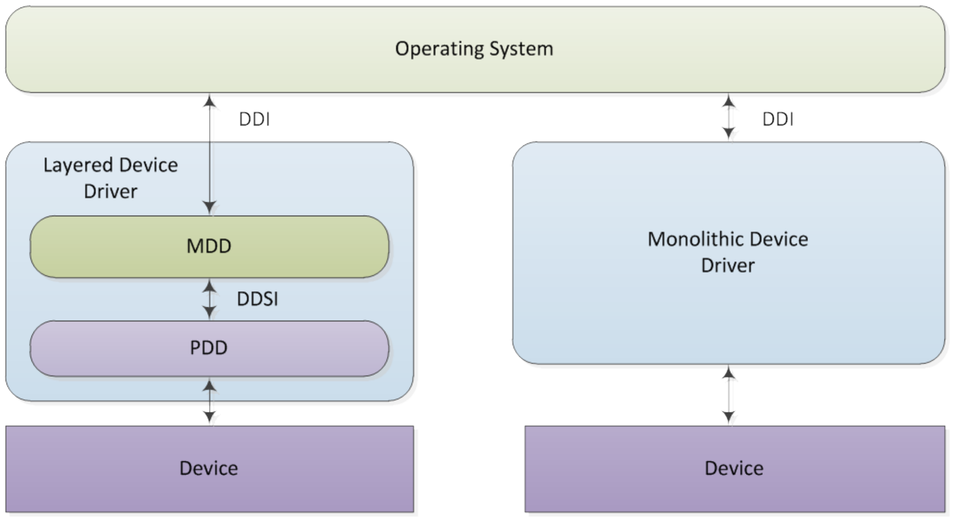

Figure 2 - Layered and monolithic device drivers

Typically, each layer is packaged in a separate DLL. Monolithic drivers, on the other hand, contain all device driver functionality in a single DLL. Figure 2 illustrates how these components interface the operating system to device hardware.

The OS accesses each device driver type through a common device driver interface (DDI) so that layered drivers and monolithic drivers can be used interchangeably, without the OS having to know what type of architecture the device driver uses. The layered architecture defines an additional interface between the PDD and MDD layers, while the monolithic device driver architecture defines no explicit interface between its upper interface and the lower hardware-specific code.

Each device driver architecture type offers distinct advantages. Monolithic drivers offer performance and resource-savings advantages while layered drivers offer advantages in terms of development costs, code reuse, and ease of updating. The layered device driver model simplifies the development and the process of porting device drivers. In the layered model, the developer only has to implement the PDD layer, reusing a common MDD layer that is already implemented for that class of hardware. For each device type that supports this layered architecture, Windows Embedded Compact 7 includes an MDD implementation along with an implemented device driver for that device class. Many of the provided sample drivers use this layered organization because it reduces the amount of code that developers must write when porting the sample drivers to new devices.

Layered Drivers

In the layered driver model, the MDD layer implements a set of functions and I/O control (IOCTL) codes that is common for a certain class of device drivers. This upper interface of the device driver MDD defines the DDI for that class of driver. The MDD layer also implements an interrupt service thread (IST) and defines the interface for interacting with the lower PDD layer of the driver. This lower interface between the MDD and PDD is called the device driver service interface (DDSI). The PDD layer implements the functionality that is specific to a particular hardware device. In effect, the PDD maps the functionality provided by a particular hardware device to the service interface presented by a particular MDD.

Distinctions between these two layers are described in the following table.

Table 5 - Device Driver Layers

Layer	Characteristics
MDD (upper layer)	• Accesses its devices indirectly, through the PDD layer. • Common to all platforms and functions both as source code and as a library. • Links to the PDD layer and uses the DDSI functions implemented in the PDD layer. • Implements a Device Driver Interface (DDI), a set of functions called by the applications or other operating system components such as GWES.

Layer	**Characteristics**
PDD (lower layer)	• Hardware dependent; typically must be customized and ported to specific hardware. • Interfaces with an MDD and hardware. • Implements a Device Driver Service Interface (DDSI), a set of functions called by the MDD layer.

The MDD can present a stream interface or a native interface. You can use a given MDD in multiple device drivers, each with a different PDD, so that different hardware devices of the same class are accessible through the same interface. For example, multiple device drivers that share a single MDD implementation can manage multiple sound devices, each with a corresponding PDD. This makes it possible for an audio mixer application to change the volume in the same way on each device, regardless of any differences in the underlying hardware.

Microsoft provides sample PDD and MDD implementations for a range of devices and device types. Keep in mind that if you modify an MDD layer, you are responsible for maintaining those changes when Microsoft releases newer versions of that MDD.

When to Create a Layered Driver

The layered driver model is a good choice if any of the following are true:

- You want to accelerate your development time by reusing an existing MDD for your device driver, concentrating your efforts on the hardware-specific PDD.
- You want to reuse source code from an existing layered device driver.
- You want to save time and effort during the development of driver updates, such as providing quick fix engineering (QFE) fixes to customers. Restricting modifications to the PDD layer increases development efficiency.

Most Windows Embedded Compact 7 device drivers are layered device drivers so that you can reuse this provided device driver functionality. When you are porting a device driver to new hardware, you generally do not need to modify the code in the MDD layer, because it contains code that is common to all drivers of that class. For example, audio drivers are made up of a PDD component and an MDD component, and the PDD is the only component that you must change in order to support a new sound device.

For More Information About Layered Drivers

If you are creating a layered driver, likely you are developing a new PDD and re-using an existing MDD that corresponds to your device class. The following Windows Embedded Compact 7 Documentation explains each of the layered device classes in detail:

Table 6 - Layered Driver Topics

Topic	Description
Accelerometer Driver	Describes the MDD and PDD functions of device drivers managing hardware that measures acceleration caused by gravity or other external forces.
Audio Drivers	Describes the MDD waveform audio and mixer functions as well as the PDD functions of device drivers that manage sound playback devices.
Battery Drivers	Describes the PDD functions and IOCTLs of battery drivers that report battery status to the OS.
Camera Drivers	Describes the MDD and PDD functions of device drivers that manage video and still image capture devices.
Flash Drivers	Describes the MDD and PDD functions of device drivers that manage flash memory devices.
Keyboard and Mouse Drivers	Describes the DDI data types and structures as well as the MDD and PDD functions of keyboard and mouse device drivers.
Remote NDIS Drivers	Describes the MDD and PDD functions of device drivers that provide functionality for a device to act as a host computer's miniport driver across an I/O bus such as USB.
Serial Port Drivers	Describes the MDD functions, PDD functions, and IOCTLs for device drivers that manage serial port devices.
USB Function Controller Drivers	Describes the MDD and PDD functions for USB function controller device drivers.
USB Host Controller Drivers	Describes the MDD and PDD functions for UHCI, OHCI, and EHCI USB host controller device drivers.

Each device driver topic listed in this table contains detailed information about the MDD and PDD functions implemented by that class of device driver. If you are implementing a new PDD for a hardware device, you implement the device driver PDD functions (DDSI) documented for that driver type.

Monolithic Drivers

A monolithic driver contains the functionality for both the logic to control the hardware and the interface to the operating system and applications in a single DLL. A monolithic driver accesses its hardware device without using an intermediate PDD layer, and it exposes the functionality of the device directly to the operating system.

Device driver developers typically use monolithic drivers in situations where performance and efficiency are critical factors, because it avoids the overhead associated with the function calls that take place between the MDD and PDD components of a layered driver. A monolithic driver can actually be simpler, more efficient, and easier to implement if the functionality of the target device corresponds closely to the semantics of the DDI functions that it exposes to applications. However, because monolithic drivers rely less on code reuse and do not leverage shared interface routines available in an MDD, monolithic drivers are, in general, more complex to develop than layered drivers.

When to Create a Monolithic Driver

The monolithic driver model is a good choice if any of the following are true:

- You are developing a device driver for uncommon, custom hardware and there is no existing MDD that you can reuse.
- You are reusing source code from an existing monolithic device driver.
- You need the additional performance that you can gain by avoiding additional function calls between separate layers in the driver architecture.
- You need the slight memory savings that is possible in a monolithic driver over a layered driver for the same device.
- The capabilities of your device are well-matched to the functions in the stream or native interface of your driver.

For More Information About Monolithic Drivers

If you are creating a monolithic driver, you are likely developing a driver that is similar to one of the existing monolithic device drivers in Windows Embedded Compact 7. The following Windows Embedded Compact 7 Documentation explains each of the layered device classes in detail:

Table 7 - Monolithic Driver Topics

Topic	Description
Display Drivers	Describes the graphics device interface (GDI) functions and other display functions implemented by monolithic display drivers.
Block Drivers	Describes the IOCTL interface presented by the CD-ROM, Disk, and other block storage device drivers.
HID Drivers and HID Parsers	Describes the functions, structures, and macros used by Human Interface Device (HID) drivers and parsers.
Network Drivers	Describes the network driver interface specification (NDIS) core and general-use functions exposed by network device drivers.
Notification LED Drivers	Describes the functions exposed by device drivers that manage hardware LEDs.
Printer Drivers	Describes the printer driver and printer port monitor functions exposed by printer device drivers.

Topic	Description
PCI Bus Drivers	Describes the structures and functions exposed by PCI Bus Drivers.
Secure Digital (SD) Card Drivers	Describes the structures, status codes, registry settings, macros, IOCTLs, and functions exposed by SD card device drivers.
Smart Card Drivers	Describes the structures, registry settings, IOCTLs, and functions exposed by smart card drivers.

Sample code for most of these drivers is available at %_WINCEROOT%\public\common\oak\drivers.

Kernel-Mode and User-Mode Drivers

Your device driver can run in kernel memory space (*kernel-mode driver*) or in a specialized user-mode host process (*user-mode driver*). User-mode drivers offer the advantage that a driver failure in user mode only affects the current process, whereas the failure of a kernel-mode driver can impair the entire operating system. Kernel-mode drivers are generally more efficient than user-mode drivers. In kernel mode, drivers have full access to the hardware and kernel memory, although function calls are generally limited to kernel APIs. Device drivers running in user mode do not have direct access to kernel memory or to kernel APIs. The drivers loaded by the GWES and the file system can only be kernel-mode drivers. The Device Manager loads both kernel-mode and user-mode drivers, but you can't run native drivers in user-mode—user-mode is restricted to stream drivers only. Each driver type can be either layered or monolithic in structure.

Kernel-Mode Drivers

Windows Embedded Compact 7 runs device drivers in kernel mode, by default. Kernel-mode drivers have direct access to user memory buffers, so kernel-mode drivers incur less overhead than user-mode drivers in moving data to and from the device. User-mode drivers require additional resource overhead for marshaling user data across process boundaries. For more information on data marshaling, see the section "Marshaling Data between Memory Buffers" in the companion article, *Implementing Your Device Driver* (http://go.microsoft.com/fwlink/?LinkID=210237).

Because kernel-mode drivers have direct access to both kernel memory and user memory, a low-quality kernel-mode driver can result in overall system instability, so kernel-mode drivers must be robust. A kernel-mode driver error can cause a kernel error, which results in system failure. If you are developing a kernel-mode driver, you must check and handle all inputs to driver functions carefully, and you must assume that these inputs come from untrusted sources.

When to Create a Kernel-Mode Driver

The kernel-mode driver model is a good choice if any of the following are true:

- Your device driver is a network driver, a file system driver, or it interfaces with the GWES. These subsystems only work with kernel-mode device drivers.

- Your device driver is a native driver. Only stream drivers can run in user-mode.
- The efficiency of your device driver is more important than possible system instability due to a driver error.
- You require access to internal kernel structures or kernel APIs in order to control your device.
- You require direct, synchronous access to user memory buffers.

For More Information About Kernel-Mode Drivers

To learn more about kernel-mode drivers, see the following in Windows Embedded Compact 7 Documentation.

Table 8 - Kernel-Mode Driver Topics

Topic	Description
Ceddk.dll	Describes functions used by drivers for handling bus address translations, allocating and mapping device memory, setting up direct memory access (DMA) buffers, and performing I/O operations.
Kernel Functions	Describes functions for allocating physical memory, mapping between virtual and physical memory addresses, forwarding device I/O controls, interrupt handler registration, and interrupt masking.

User-Mode Drivers

User-mode drivers can improve system stability, fault-tolerance, and security. Because a user-mode driver can be isolated from the kernel and other user-mode drivers, it can be reloaded if it fails without restarting the operating system. However, this does not mean that you can omit security and stability requirements when developing a user-mode driver.

Unlike kernel-mode drivers, user-mode drivers do not have access to kernel memory or kernel APIs, and there are performance penalties when running in user mode. User-mode drivers have only limited access to user memory buffers, and they cannot directly access device hardware. To communicate with underlying hardware and perform useful tasks, user-mode drivers must be able to access system memory and privileged APIs unavailable to standard user-mode processes. To facilitate this, user mode drivers rely on two system components:

- The *User Mode Driver Reflector*, which resides inside the Device Manager.
- The *User Mode Driver Host*, which is a user-mode application that is launched and managed by the User Mode Driver Reflector.

More than one user-mode device driver can run inside the User Mode Driver Host, and user-mode drivers can be separated from one another by being loaded into different host processes. When your driver is flagged as a user-mode driver in the registry, the Device Manager connects it to the User Mode Driver Reflector. The User Mode Driver Reflector launches the appropriate User Mode Driver Host process and forwards I/O requests to it. The User Mode Driver Host process in turn forwards I/O requests to your user-mode driver. This relationship is shown in the following figure.

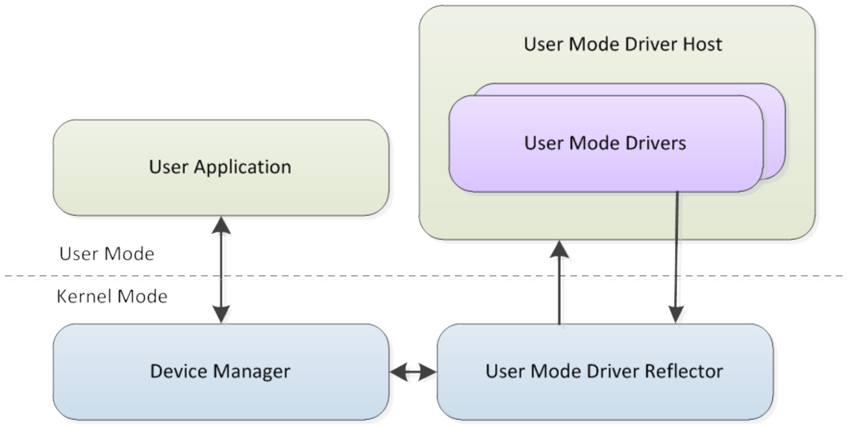

Figure 3 - User mode driver framework

The User Mode Driver Reflector runs in kernel mode, translates memory addresses of the calling process to the driver's address space (also known as buffer marshaling), and calls privileged memory management APIs on behalf of user-mode drivers. The Reflector also validates input parameters before performing a driver's requested action by checking the registry settings for that device driver to determine whether it can perform the requested action.

Because applications do not differentiate between user-mode and kernel-mode drivers, the User Mode Driver Reflector is responsible for masking the differences between user-mode drivers and kernel-mode drivers from the rest of the system. If your driver does not use kernel APIs that are unavailable in user mode, your driver can run in either kernel mode or user mode through the Reflector—the Reflector is transparent so that your driver can work the same way, either in either kernel mode or user mode, without modifications.

To configure your driver to run as a user-mode driver, you must configure registry settings to specify that your driver can run in a user-mode driver host process, and you specify the name of the process that your driver must run in (such as Udevice.exe). For more information on configuring the registry to run your device driver in user-mode, see User Mode Driver Framework Registry Settings (http://go.microsoft.com/fwlink/?LinkID=210292) in Windows Embedded Compact 7 Documentation.

When to Create a User-Mode Driver

The user-mode driver model is a good choice if any of the following are true:

- Your system must be secure from a possibly compromised device driver.
- You are willing to incur minor performance losses so that the system can recover more gracefully from a driver error.
- Your driver must be able to display directly to a user interface.
- You are developing your device driver by reusing code from an existing user-mode driver.
- You want to initially develop your device driver in user mode (for ease of development) so that you can later adapt it for kernel mode.

For More Information About User-Mode Drivers

To learn more about user-mode drivers, see the following in Windows Embedded Compact 7 Documentation.

Table 9 - User-Mode Driver Topics

Topic	Description
User Mode Driver Framework	Describes functions and configuration settings for loading and running a device driver in user mode.
Memory Management Functions	Describes user-mode functions for allocating and deallocating memory, obtaining information about the heap or physical memory and virtual memory of the system, and changing memory properties.

Bus Drivers

A *bus driver* is a device driver that controls a bus. Bus drivers can have one or more of the following responsibilities:

- Manage physical buses such as PC Card, USB, or PCI.
- Load drivers on a physical bus for hardware that the bus driver does not directly manage. An example is the Bus Enumerator, which is the root bus driver that loads built-in drivers, if the platform has PCI bus support.
- Load device drivers that manage hardware indirectly through another device driver.

Bus drivers read configuration information about a bus and “walk” buses to discover and enumerate devices on the bus. Bus drivers provide the Device Manager with information to create device handles and bus-relative names. They also provide enough information to identify themselves to drivers and applications for bus control operations.

Bus drivers manage the **\$bus** namespace and create *bus names* for bus-based devices. Bus names relate to the underlying bus structure and are usually derived from the bus type, the bus number, and other specific identifiers such as the device number and function number. The **\$bus** namespace provides a way to perform operations on a device that are different from typical device operations. Because these operations typically go through the bus driver, the bus driver can provide additional security for your system. See the section “Bus-based Driver Names” later in this document for more about bus names and the **\$bus** namespace.

Because Windows Embedded Compact 7 includes bus drivers for most popular bus topologies, you typically don’t develop bus drivers—instead, you develop your device driver to work with an existing bus driver if your device is on a bus. The Windows Embedded Compact 7 PCI bus driver walks and enumerates devices on an attached PCI bus. The Windows Embedded Compact 7 USB bus driver connects a USB hub to your target platform and exposes the USB driver interface functions listed in the Universal Serial Bus Specification.

For More Information About Bus Drivers

To learn more about bus drivers, see the following in Windows Embedded Compact 7 Documentation.

Table 10 - Bus Driver Topics

Topic	Description
PCI Bus Drivers	Describes PCI bus driver device configuration functions, structures, and IOCTLs.
USB Bus Driver	Describes USB host controller drivers that control USB hubs connected to a Windows Embedded Compact device.

Bus-Agnostic Drivers

A *bus-agnostic driver* can execute on different buses. For example, a bus-agnostic driver can execute on a PCI bus, a 16-bit PC Card bus, or a bus that is specific to a hardware platform. Typically, you can more easily migrate bus-agnostic drivers between platforms than other types of drivers.

A bus-agnostic driver does not call functions that are specific to a hardware platform. Each bus-agnostic driver gets its resources from the registry, requests configuration information from its parent bus driver, sets power states through its parent bus driver, and translates bus addresses to system addresses through its parent bus driver.

A bus-agnostic driver is a standard driver that is relatively simple to implement, and which can be shared by various implementations for the same hardware chipset. For example, the bus-agnostic PCI NE2000 driver works correctly for the NE2000 PCMCIA card. The NE2000 PCMCIA card functions the same with the PCI NE2000 driver as it does with the PCMCIA NE2000 driver.

Selecting a Device Driver Loading Strategy

Depending on when your device driver is needed by the operating system, your device driver can be loaded in one of two ways:

1. Automatically during system start up (statically).
2. On demand after system startup (dynamically).

For example, if you are implementing a block driver for the main file system, you will likely want to load and start it during system startup so that the OS can access the file system to load applications. If this block driver is used only for mounting and accessing a removable secure digital (SD) card, then it makes sense to defer the loading and starting of this driver until the user inserts an SD card, so that you can conserve memory usage.

Depending on the kind of driver you implement, your driver is loaded and started by one of three processes, as shown in the following table.

Table 11 - Device Driver Loading Processes

Process	Loads
Device Manager	Stream drivers.
Graphics, Windowing, and Events Subsystem (GWES)	Keyboard, video adapter, touch screen, printer, and mouse native drivers.
File System	Native file system drivers.

In each case, the process that loads your device driver looks at registry settings to determine how to load and start your driver. To configure the OS to load and start your device driver, you select the appropriate process for loading your device driver type, and configure the registry settings for that process with information about how and when to load your device driver. The following sections explain this process in more detail.

The Device Manager

The Device Manager manages the stream device drivers on the system; it loads, unloads, and interfaces with stream drivers as shown in Figure 4. The Device Manager consists of two DLLs that are loaded by the kernel: Device.dll and Devmgr.dll. The kernel loads Device.dll, which is a thin shell that loads Devmgr.dll, and Devmgr.dll implements the core Device Manager functionality. Because the Device Manager is implemented as two DLLs, your device driver can link directly with the Device Manager, and invoke Device Manager functions without incurring the overhead of a system call.

During the boot process, the OEM Adaptation Layer (OAL) loads the kernel, and the kernel loads the Device Manager. Once the Device Manager starts, it runs continuously, loading and unloading device drivers, tracking loaded device drivers and their interfaces, notifying the kernel when it loads or unloads a stream device driver, and notifying the user when device interfaces become available and unavailable. (See the section “Supporting Device Interfaces” in the companion article *Implementing Your Device Driver* for more on device interfaces.) The Device Manager also sends power notification callbacks to device drivers and notifies the kernel when it discovers a device interface that supports file operations (such as **CreateFile**).

When a user application calls **CreateFile** to create a handle to your device driver and then calls file I/O functions like **ReadFile** and **WriteFile** on this handle, the file system routes these calls to corresponding Device Manager functions like **XXX_Read** and **XXX_Write**. In turn, the Device Manager calls the corresponding entry points in your device driver, such as **XXX_Read** and **XXX_Write**. Similarly, the file system routes all **IOControl** operations on that handle to the Device Manager, and the Device Manager forwards these calls by calling the **XXX_IOControl** entry point in your device driver.

Note that neither the file system nor the user application calls your driver’s **XXX_** functions directly; instead, the file system routes each of these calls through the Device Manager, as shown in the following figure.

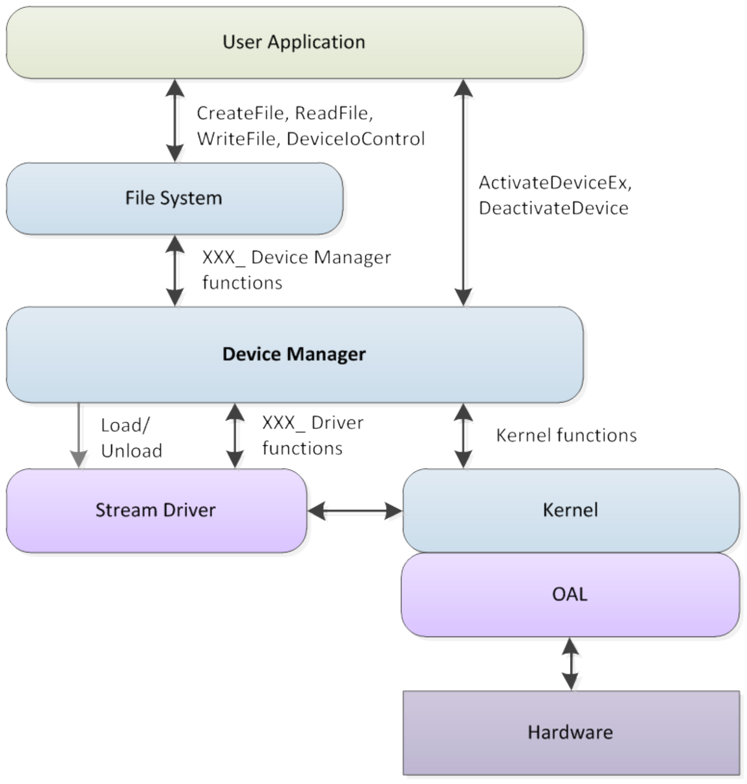

Figure 4 - Device Manager

You can configure the Device Manager to load your driver automatically at boot time by configuring registry settings specific to your device driver, or you can load your driver dynamically by calling the Device Manager's **ActivateDeviceEx** function from an application. In either case, the Device Manager loads your device driver using the same registry flags and settings. If your driver is a user-mode driver, the call to **ActivateDeviceEx** will result in the Device Manager calling the User Mode Driver Reflector as well.

For more about **ActivateDeviceEx**, see ActivateDeviceEx (http://go.microsoft.com/fwlink/?LinkID=210241) in Windows Embedded Compact 7 Documentation. For more about the Device Manager, see Device Manager (http://go.microsoft.com/fwlink/?LinkID=204667) in Windows Embedded Compact 7 Documentation.

Registry Settings

If you are loading your device driver automatically, you must configure your device driver's settings in a subkey under the Device Manager's *RootKey*, which, by default is set to **HKEY_LOCAL_MACHINE\Drivers\BuiltIn**. For example, to load "MyDriver" automatically at boot time, create this subkey:

HKEY_LOCAL_MACHINE\Drivers\BuiltIn\MyDriver

If you are loading your device driver dynamically, you can configure your registry settings in any location you like, for example:

HKEY_LOCAL_MACHINE\MyDriver

The registry path that you select is what you pass to **ActivateDeviceEx** to load your driver. The following table lists the settings that you configure at this registry location.

Table 12 - Registry Settings for Loading Stream Drivers

Setting	Description
Dll	Required. Specifies the driver file name.
Prefix	Optional. Defines the prefix of the stream driver and part of the device name for accessing the driver through the file system. It must match the prefix that is used for implementing driver functions if the DEVFLAGS_NAKEDENTRIES bit of the Flags field is not set (see the Flags field, below). DEVFLAGS_NAKEDENTRIES means that driver functions were implemented without a prefix (for example, Init, Deinit, Write, and so on).
Order	Optional. Determines the load order of the device driver. The device manager loads device drivers in the order specified by this parameter. If you do not declare this parameter, your driver will be loaded after all drivers that declare this parameter. You can use this parameter to arrange dependencies between automatically-loaded drivers during system start.
Index	Optional. The index is part of the device name for accessing your driver through the file system. The index is appended to the right of the prefix (above) to create a unique driver name (see the section “Selecting a Device Driver Access Method” later in this document). If you do not specify this parameter, the Device Manager will automatically assign sequential index values for each new device instance.
IClass	Optional. Specifies the device class, which is a GUID that refers to a device interface. For more on IClass registry subkeys, see Device Interface Notifications (http://go.microsoft.com/fwlink/?LinkID=210251) in Windows Embedded Compact 7 Documentation.
Flags	Optional. Specifies how the Device Manager loads the driver. See ActivateDeviceEx (http://go.microsoft.com/fwlink/?LinkID=210241) in the Windows Embedded Compact 7 Documentation for the flag values that you can set.

Windows Embedded Compact 7 uses the following procedure to load device drivers that are configured to be loaded automatically:

1. The Device Manager reads the **HKEY_LOCAL_MACHINE\Drivers\BuiltIn** registry key to determine the list of drivers to load.
2. The Device Manager calls **ActivateDeviceEx** to load the driver specified in the **Dll** setting of the registry key **HKEY_LOCAL_MACHINE\Drivers\BuiltIn**. By default, this driver is the bus enumerator (BusEnum.dll).
3. The bus enumerator scans **HKEY_LOCAL_MACHINE\Drivers\BuiltIn** for subkeys that refer to additional buses and devices, examining the **Order** value in each subkey to determine the load order of each corresponding device driver.
4. Starting with the lowest **Order** values, the bus enumerator steps through each subkey and calls **ActivateDeviceEx**, passing in the registry path for that driver.

5. Each call to **ActivateDeviceEx** causes the Device Manager to load the device driver file specified by the **Dll** value in the registry path passed to **ActivateDeviceEx**.
6. The Device Manager creates an additional subkey under **HKEY_LOCAL_MACHINE\Drivers\Active** to keep track of the loaded device driver.

The Device Manager controls the **Active** key; only the Device Manager can directly access this key for read or write access. You can indirectly access the **Active** key through the *pContext* parameter that is passed to your device driver when your **XXX_Init** entry point is called. For more information about **XXX_Init**, see XXX_Init (Device Manager) (http://go.microsoft.com/fwlink/?LinkID=210295) in Windows Embedded Compact 7 Documentation. For more information about the Active registry key, see Device Manager Registry Keys (http://go.microsoft.com/fwlink/?LinkID=210253) in Windows Embedded Compact 7 Documentation.

Graphics, Windowing, and Events Subsystem

The Graphics, Windowing, and Events Subsystem loads user interface device drivers that are used exclusively by GWES:

- Display drivers
- Keyboard drivers
- Mouse drivers
- Touch screen drivers
- Printer drivers

These device drivers are native drivers, and each class of device shown in the preceding list has its own interface with GWES. GWES loads each driver based on a registry key for that device driver class. Depending on which class of user interface driver you implement, the details for configuring the registry for loading your device driver can be found in Windows Embedded Compact 7 Documentation:

Table 13 - Topics by GWES Driver Type

Driver Type	Reference Topic in Windows Embedded Compact 7 Documentation
Display	Display Driver Registry Settings
Keyboard	Keyboard and Mouse Driver Registry Settings
Mouse	Keyboard and Mouse Driver Registry Settings
Touch Screen	Touch Screen Driver Registry Settings
Printer	Printer Driver Registry Settings

For example, GWES loads the display driver DLL specified in the registry key **HKEY_LOCAL_MACHINE\System\GDI\Drivers\Display**. To load your display driver, you configure the file name of your device driver in the “Dll” setting of this registry key, as described in Display Driver Registry Settings (http://go.microsoft.com/fwlink/?LinkID=210254) in Windows Embedded Compact 7 Documentation.

File System

The File System (FileSys.dll) calls the Storage Manager to load drivers that specify a particular file system, such as FATFS or UDFS. See Storage Device Settings (http://go.microsoft.com/fwlink/?LinkID=210284) in Windows Embedded Compact 7 Documentation for details about configuring registry settings for loading file system drivers.

Selecting a Device Driver Access Method

If you are creating a native driver, the access method for your device driver is built into the native interface that you expose to GWES or the file system. For example, if your device driver is a display driver, GWES uses the **GPEEnableDriver** function of your driver to connect your driver to GWES. Unlike stream drivers, native drivers are normally opened for exclusive access by the subsystem that controls that driver. Therefore, the access method is chosen for you, if you are creating a native driver.

If you are creating a stream driver, however, the OS and applications can gain access to your device driver through the file system by making a call to **CreateFile** to get a handle to your device. After an application obtains this handle, it can call **ReadFile** and **WriteFile** to perform read and write operations, which the Device Manager translates into corresponding calls to stream interface functions on your device driver. So that Windows Embedded Compact 7 can recognize stream device resources and redirect file operations to the appropriate stream driver, it uses special naming conventions for file names that represent device resources, rather than ordinary files. Your stream device driver can use one or more of these three device naming conventions: *legacy*, *device-based*, and *bus-based*.

Legacy Driver Names

The legacy driver namespace, first used in earlier versions of Windows CE, constructs each device name from a device prefix and an index. Legacy names use the following format:

XXX[0-9]:

XXX is the three-letter prefix that represents the driver name, followed by a digit between zero and nine (inclusive) that specifies an instance index, and ending with a colon. For example, the first port on the **COM** serial communications driver is managed through instance index 1, which results in the name **COM1:**, while the second port is accessed through the name **COM2:**. An application opens the stream driver for the second serial port by passing its name to **CreateFile**:

```
CreateFile(L”COM2:”, …)
```

Because the index is restricted to a single digit, only 10 device names with the same driver prefix can be accessed through the legacy driver namespace. The first driver instance corresponds to index 1, the second instance corresponds to index 2, and the ninth instance corresponds to index 9. However, the tenth instance corresponds to index 0.

Here are some examples of legacy driver names that are currently in use:

COM	Serial driver
LPT	Printer port
CON	Console driver
DSK	Storage driver
ACM	Audio compression manager
WAV	Audio wave driver

Device-based Driver Names

The device namespace is similar to the legacy namespace but it supports driver instance index values with multiple digits. Device names are constructed by adding the device namespace **\$device** to the device prefix and the instance index. Unlike legacy device names, this instance index value can be greater than nine. Device names use the following format:

\$device\XXX*[index]*

XXX is the three-letter prefix that represents the driver name, followed by one or more digits that specify an index. Note that device-based names do not use a terminating colon.

For example, the twelfth port on the **COM** serial driver has the device name **\$device\COM12**. If the first storage device is accessed through the name **\$device\DSK1**, an application opens the stream driver for this storage device by passing its name to **CreateFile**:

```
CreateFile(L”\$device\DSK1”, …)
```

Because the index is not restricted to a single digit, more than 10 device names with the same driver prefix can be accessed through the device-based driver namespace.

Bus-based Driver Names

If your device driver is a stream driver that manages hardware on a bus such as PCMCIA or USB, applications access your device through a *bus name*. Bus names provide a way to perform operations on a device that are different from normal device operations such as read and write. Applications and other device drivers communicate with your driver’s parent bus driver for services such as loading, unloading, and power management.

Bus names derive from the bus type, bus number, and other bus-specific identifiers such as device number and function number. This bus naming convention assumes that all buses of a particular type are uniquely numbered. When this combination of identifiers is unique, no name collisions occur.

Each bus name begins with the prefix **\$bus** and uses the following format:

\$bus\<*bus_name*>_<*bus_number*>_<*device_number*>_<*function_number*>

\$bus is the namespace identifier that indicates that this is a bus name, *bus name* refers to the name or type of the bus, and *bus number*, *device number*, and *function number* are bus-specific identifiers. For example, the device name for function 0 on the second device on the first PCI bus is **\$bus\PCI_0_1_0**. If you have a serial port that is the third function number on the first device of a single PCMCIA bus, it has the bus name **\$bus\PCMCIA_0_0_2**. An application opens the device in the first example by passing this bus name to **CreateFile**:

```
CreateFile(L”\$bus\PCI_0_1_0”, …)
```

Bus names are created by the bus driver for a particular bus. Therefore, unlike legacy or device-based names, bus names must correspond to the underlying structure of the bus. Bus drivers for different types of buses may create bus names differently, depending on how those buses are structured.

For more about bus-based driver names, see “Bus Drivers” earlier in this document.

Conclusion

This article explained the essentials of the device driver architecture in Windows Embedded Compact 7 and gave you the background information that you need to plan your device driver implementation. You learned how to characterize your driver by mapping the functionality of the underlying hardware to the interfaces that software components use when accessing your driver. You learned about bus drivers, stream drivers versus native drivers, monolithic versus layered drivers, and kernel-mode versus user-mode drivers, and how to select from these driver types an appropriate set of options that meets your requirements. You learned about the mechanisms for loading and starting your device driver (the Device Manager, GWES, and the file system), and about the various driver naming schemes that the OS and applications use to access device drivers. With this knowledge, you can now begin the implementation phase of your device driver project, as described in the next article in this series, *Implementing Your Device Driver* (http://go.microsoft.com/fwlink/?LinkID=210237).

Additional Resources

Windows Embedded website (http://go.microsoft.com/fwlink/?LinkID=203338)

Copyright